A Software Caching System for UPC

Wei Chen Jason Duell Jimmy Su
CS265 Project
May 16, 2003

Abstract

Unified Parallel C (UPC) is a parallel language that uses a Single Program Multiple Data (SPMD) model of parallelism within a global address space. One unique feature of UPC is that it allows a separate consistency protocol for each shared variable, which can be either "strict" or "relaxed", with the goal that "strict" provides sequential consistency while "relaxed" provides better performance. The current specification of the UPC memory consistency model, however, places severe restrictions on a conforming implementation ability to reorder relaxed accesses. To solve this problem, We have proposed a new UPC memory model that allows compilers to aggressively optimize for relaxed accesses. In particular, this enables us to build a software-based caching mechanism that can provide a significant performance boost to fine-grained UPC applications written in the shared memory programming style. A prototype caching system is currently operational in the Berkeley UPC compiler, and we evaluate its implementation and present preliminary performance results for a simple matrix multiplication benchmark. To further bridge the gap between fine-grained and coarse-grained applications, we also discuss compiler techniques that could automatically transforms array accesses in loops into equivalent bulk prefetching operations.

1 Introduction

1.1 Unified Parallel C

UPC (Unified Parallel C) [11] is a parallel extension of the C programming language aimed at supporting high performance scientific applications. The language adopts the SPMD programming model, so that every thread runs the same program but keeps its own private local data; each thread has a unique integer identity expressed as the `MYTHREAD` variables, and the `THREADS` variable represents the total number of threads, which can either be a compile-time constant or specified at runtime. In addition to each thread's private address space, UPC provides a shared memory area to facilitate communication among threads, and programmers can declare a shared object by specifying the `shared` type qualifier. While a private object may only be accessed by its owner thread, all threads can read or write to data in the shared address space. Because the shared memory space is logically divided among all threads, from a thread's perspective the shared space can be further divided into a local shared memory and remote one. Data located in a thread's local shared space are said to have "affinity" with the thread, and compilers can utilize this affinity information to exploit data locality in applications to reduce communication overhead. Figure 1 illustrates the UPC memory model.

As its memory model suggests, pointers in UPC can be classified based on the locations of the pointers themselves and of the objects they point to. Accesses to the private area behave identically to regular C pointer operations, while accesses to shared data are made through a special pointer-to-shared construct. The speed of local shared memory accesses will be lower than that of private accesses due to the extra overhead of determining affinity, and remote accesses in turn are typically significantly slower because of the network overhead. Figure 2 illustrates three different kind of UPC pointers: private pointers pointing to objects in the thread's own private space (P1 in the figure), private pointers pointing to the shared address space (P2), and pointers living in shared space that also point to shared objects (P3).

1.2 The Need for Caching in UPC

One characteristic of parallel programs in distributed memory environment is that generally a significant performance gap exists between local and remote shared memory accesses. While the overhead of local memory operations is usually no more

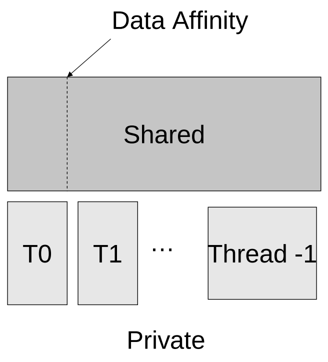
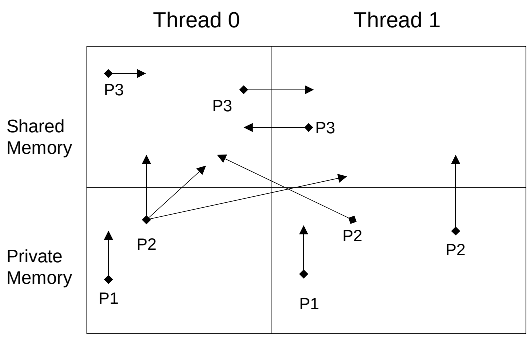


Figure 1. the UPC memory model.



P1 == private pointer
P2 == pointer-to-shared
P3 == shared pointer-to-shared

Figure 2. Types of pointers in UPC

than a dozen cycles, a recent study [5] shows that the latencies for remote accesses in current high-performance networks are in the order of microseconds, which translates into thousands of cycles for modern processors. Global address space languages like UPC are liable to suffer from this performance gap, as they encourage a programming style that favors small message traffic. As a result, a runtime caching mechanism is important for reducing communication latencies caused by eliminating redundant accesses in the program that compilers have not identified due to conservative alias and data dependence analysis.

A caching system is especially helpful for fine-grained programs, since it can provide data prefetching by reading an entire cache block in one transfer instead of performing multiple small reads. Coarse-grained applications that use bulk memory transfer for communication are less likely to benefit from caching, as the bulk access functions can be thought of as providing caching functionality at the program level [10]. UPC programmers can thus use an "incremental" programming style, in which serial applications can initially be ported relatively naively to UPC, and then performance-critical sections of the program can be optimized by hand to use bulk data transfers (and local pointer accesses) in places where the UPC compiler has been unable to optimize the access pattern automatically. By making fine-grained accesses more efficient, caching has the potential to make fewer parts of an application require such compiler- or hand-generated optimizations in order to achieve acceptable performance. Caching can thus be used to effectively bridge the gap in performance between the coarse-grained and fine-grained programming model, allowing UPC to exhibit less of a tradeoff between performance and programmer effort.

1.3 A Software Caching Mechanism for the Berkeley UPC Compiler

To evaluate the benefits of caching for fine-grained UPC programs, we have implemented a caching system in the Berkeley UPC Compiler [7], a high-performance and portable implementation of UPC. While hardware-supported shared memory using a directory-based protocol will outperform a software-based caching implementation [8], such hardware-coherent systems are expensive to build and unlikely to scale for large number of processors. Furthermore, since portability is a major goal of the Berkeley UPC compiler, it needs to support many distributed memory systems where architecture caching support is not available. As a result, a software-only approach is adopted for our cache implementation, and this paper presents the design and preliminary performance of our prototype caching system.

The rest of the paper is organized as the follows. In section 2, we propose a new UPC memory consistency model that allows for a more efficient software-controlled cache implementation. Section 3 describes the design of our caching mechanism, and Section 4 discusses several optimizations that we plan to incorporate into the caching system. Section 5 presents the preliminary performance results, while Section 7 concludes the paper.

2 A New Memory Consistency Model for UPC

2.1 The UPC Memory Consistency Model

When writing parallel code most programmers intuitively rely on the notion of sequential consistency [19], which states that a parallel execution must behave as if it's an interleaving of the serial executions by each processor, with each processor's execution sequence preserving the program order. Sequential consistency offers good programmability because it is a natural extension of the uniprocessor execution model; a parallel execution is sequentially consistent if and only if its output is identical to the execution in which each thread strictly follows the program order. It is well known, however, that standard compiler transformations for sequential programs will not preserve sequential consistency when applied in a parallel environment. In particular, Midkiff and Padua [22] demonstrated 11 scenarios where compiler optimizations cause unexpected behaviors in parallel programs due to violations of sequential consistency.

While sequential consistency offers the most intuitive view of the behavior of the memory, it is also generally considered too expensive to maintain for parallel programs. The naive method for enforcing sequential consistency is to simply require that all memory accesses from a single thread must be issued in program order, by placing fence instructions between every consecutive accesses. This approach unfortunately would incur a significant performance penalty, as it excludes most useful compiler optimizations such as code motion, prefetching, and register allocation. In particular, a caching system will no longer be functional, as caching a remote value has the implicit effects of moving all later accesses to the same shared address up to the cache miss point, resulting in reordered memory operations. Faced with this challenge, several studies ([17, 20, 25]) have proposed sophisticated algorithms that attempt to minimize the number of memory fences necessary to guarantee sequential consistency. A common weakness of such algorithms, however, is that their success hinges on the quality of aliasing and data dependence information available at compile time, and current analyses for them are generally either over-conservative or prohibitive in terms of their running time. As a result, in practice most complex applications will not significantly benefit from the sequential consistency preserving analyses.

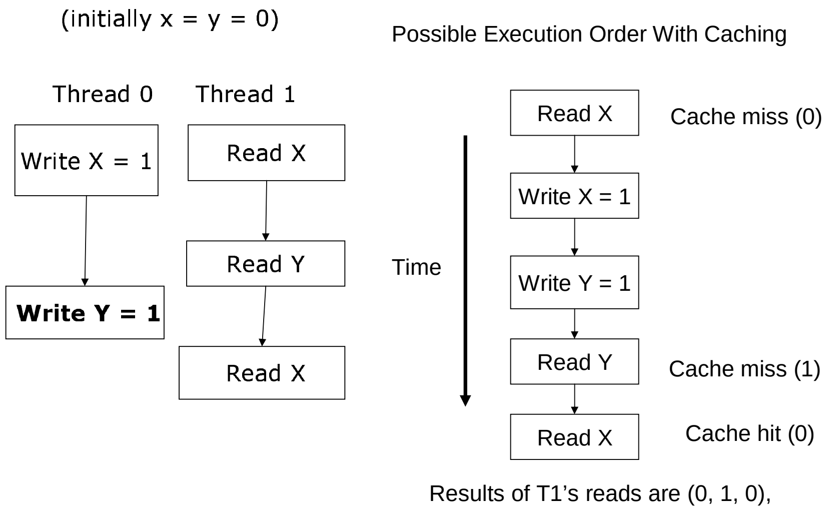
To solve the conflict between the desire for an easy-to-understand consistency model and the need of compiler optimizations to boost performance, UPC has adopted the approach of supporting two different consistency models at the language level. The insight here is that sequential consistency can be violated only if a data race exists in the program (a data race occurs when there are potentially concurrent accesses to the same memory location, and at least one of them is a write). A well written program should contain few if any data races, since their presence results in non-deterministic behavior and is thus usually a sign of buggy code.

If a program is free of race conditions, each thread can safely apply sequential compiler optimizations to its own portion of the program, provided the local data dependency requirements are still observed. Such transformations do not violate sequential consistency, because one thread's reordering operations will never be visible to others.

To support these optimizations possible for a "well-behaved" parallel program, UPC therefore provides both a strict and a relaxed memory model for variables. The programmer assures the compiler that relaxed accesses do not conflict across threads, while strict variables can be used to provide cross-thread synchronization, providing the more intuitive (and expensive) sequential consistency model across parallel threads when this is needed.

2.2 Inadequacies of the UPC 1.1 Model for Caching Remote Values

The current UPC program execution model is described in detail in the language specification [11]. Two rules in particular govern the execution of shared memory accesses: the first rule enforces local data dependency for all shared accesses on each individual threads, just as it would for ordinary loads and stores in sequential programs. The second rule maintains the program order of strict references among all threads, by disallowing any reordering of a strict reference (with respect to other shared accesses on the same thread) that could be observed by other threads. Intuitively, the specification seems to match well with the intention that compiler optimizations could freely reorder relaxed accesses while maintaining the program order among strict reference themselves as well as between strict and relaxed accesses. What the model neglects, however, is that due to the second rule strict references from one thread can inadvertently impose an ordering restriction on relaxed accesses from other threads.



Results of T1's reads are (0, 1, 0), violates UPC memory model (T1 must observe the write to X before the strict write to Y)

Figure 3. How Caching Could Violate the Current UPC Memory Model (bold letters indicate a strict access)

Consider the example shown in Figure 3, where variable y is accessed concurrently by a strict write from thread 0 and a relaxed read from thread 1. Since the UPC memory model requires that strict accesses appear in a single sequential order for all threads, if thread 1's relaxed read returns the value updated by the strict write, it must have observed all writes from thread 0 prior to the strict access. Consequently, thread 1's second read to x must reflect thread 0's earlier update, and $(0, 1, 0)$ in particular is an illegal result for thread 1's reads. This semantics, however, places a severe restriction on the caching implementation, as Figure 3 illustrates how a system that caches remote reads but otherwise maintain the program order could still unwittingly violate the current UPC memory consistency model.

To overcome this problem, each thread must be aware of the strict accesses that could be concurrently issued by remote threads. Propagating such information at runtime, however, likely will be prohibitively expensive; whenever a thread performs a strict write (or a strict read due to anti-dependencies), it must first invalidate all threads' caches to ensure that relaxed writes from all threads prior to the strict operation have been "globally performed". A write is said to be *globally performed* when all subsequent reads to the variable will either return the updated value or that of a later write. This creates a significant negative impact on program performance and also severely hinders the scalability of the caching implementation. Furthermore, this restriction on relaxed accesses is not limited to caching mechanisms alone, as any compiler optimizations that could potentially reorder memory accesses will run into the same problem. [18] presents several more scenarios where the order of relaxed accesses must be maintained as if it were under the effect of sequential consistency, due to interferences from strict accesses made by different threads.

2.3 A New Definition of the UPC Memory Model

Examples from the previous subsection illustrate that the current UPC memory consistency model is too restrictive for most important compiler optimizations. In particular, a caching mechanism can not be efficiently implemented without an excessive amount of coherence message traffic for each strict memory reference. The problem fundamentally stems from conflicting strict and relaxed accesses issued by different threads; if thread A's relaxed read obtains the value from a strict write by thread B, (indicating the strict write happens before the relaxed read), according to the current model A must also have observed all earlier shared accesses by B before the strict update. This requirement, however, can be easily satisfied if there are no data races caused by concurrent strict and relaxed accesses. Since relaxed accesses must be data-race free if the program is to appear sequentially consistent, we feel it natural to extend the notion and forbid any race conditions between strict and relaxed accesses.

With this restriction in place, the ordering constraints provided by a strict write are only guaranteed to be observed by a remote thread when that thread observed the write via a strict read. Relaxed accesses no longer have to reflect the effects of concurrent strict accesses, as this is not a condoned programming practice. We formalize this intuition into a formal specification of the UPC memory consistency model that slightly modifies the definition of weak ordering in [4]:

Defi nition 1 (The UPC synchronization model) *In a correctly written UPC program, all conflicting shared memory operations that involve relaxed accesses must be ordered by the happens-before relation for the execution.*

The *happens-before* relation, also specified in [4], defines a partial order on shared accesses based on program order as well as the synchronization operations. Note that our definition of synchronization operations include strict accesses, as they can be considered a mechanism for providing pairwise synchronizations between a thread that issues a strict write and another that performs a strict read. This also satisfies the intuition that all shared reads and writes prior to a strict operation in program order must have completed before the strict access can take place. Thus, any UPC program that obeys the above model must provide enough synchronization to guarantee that no data races involving relaxed accesses exist. The UPC Memory consistency model can now be stated in terms of this particular synchronization model:

Defi nition 2 *A UPC implementation must appear sequentially consistent to any program that obeys the synchronization model*

The new model leaves the behaviors of programs with data races involving relaxed accesses undefined, so implementations can aggressively optimize well-behaved programs without considering the degenerate cases. Furthermore, it still provides a easy to understand contract between the applications and the system, as programmers can still rely on the notion of sequential consistency as long as they follow the synchronization model. Finally the model allows for a flexible cache coherence protocol with regards to the write atomicity requirements[3]. Since the model requires that all conforming programs have no data races on relaxed accesses, when a thread issues a relaxed write, it can assume that other threads will not attempt to access the same memory location until after the thread's next synchronization operation. This allows us to experiment with various data propagation strategies discussed in [16], so long as we ensure that cache coherence is always maintained at synchronization points.

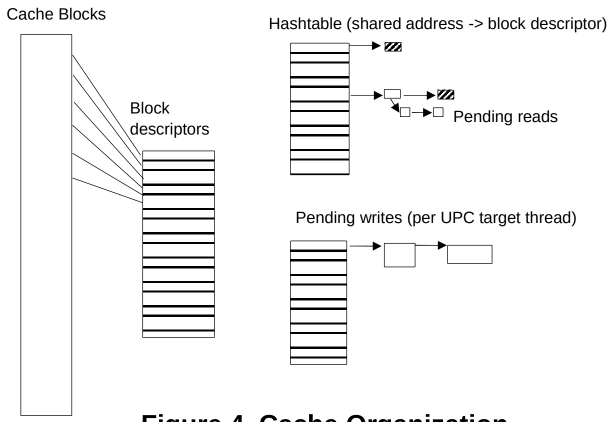

Cache Blocks
Block descriptors
Hashtable (shared address -> block descriptor)
Pending reads
Pending writes (per UPC target thread)

Figure 4. Cache Organization.

3 Basic Design

In our software caching mechanism, each thread allocates its own local cache, which is transparent to both the program and the compiler. The cache system instruments the following communication operations requested by the compiler:

Synchronous fi ne-grained accesses blocking one-sided $puts$ and $gets$ for a single variable, with semantics identical to local memory accesses (except the much higher latency).

Asynchronous fi ne-grained accesses non-blocking remote memory operations that's split into two phases: an $init\_op()$ followed by a $sync\_op()$. The initiation phase issues a non-blocking network call, while the synchronization phase guarantees the operation's completion.

Bulk accesses bulk memory transfer functions that access large chunks of consecutive shared memory locations. The accesses can be either blocking or non-blocking.

Synchronization operations barriers, locks, etc.

Currently the compiler translates all strict accesses into synchronous operations and relaxed accesses into asynchronous ones. This strategy matches well with the intuition that relaxed memory operations can be freely reordered, while the program order of strict accesses must be enforced. In the future we may relax this requirement, however, to allow for more optimization opportunities for strict accesses. Also, the cache system does not affect relaxed local accesses so it will not introduce additional overheads for them. Strict local accesses, on the other hand, still must be treated as a synchronization operation to ensure our memory consistency model is not violated. Finally our caching system is based on a “local knowledge” scheme [9], and thus requires no global communication for maintaining cache coherence.

3.1 Cache Structure

Figure 4 illustrates the structure of our cache system. Each thread allocates an array of cache blocks, with a corresponding set of block descriptors. A hashtable is used to perform cache lookup, by translating a shared address to its corresponding block descriptor. Both the size and the number of cache blocks per thread are parameters configurable by the user, with the restriction that their product is less than the total size of the shared memory space allocated by each thread.

The function of a block descriptor is to record the state that a cache block may be in:

1. invalid: There is no data in the cache block.
2. ready: The data in the cache block is valid and can be read.
3. pending: There is a pending incoming read operation that will fill the cache block.

Figure 5 shows a transition diagrams for the above three states. Note that a cache hit is possible only if the corresponding cache block is in the ready state. If the block is in the pending state, the block descriptor also includes a handle to the pending read, so that the caching system could synchronize on the operation when demanded by the compiler.

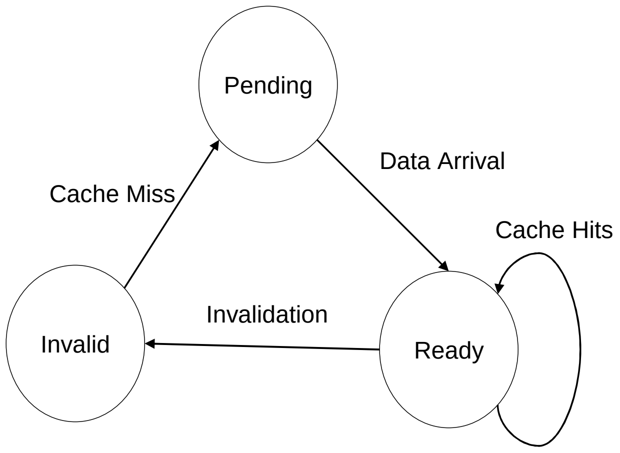


Figure 5. State Diagram of a Cache Entry

As Figure 4 also shows, a separate *update queue* is created on each thread to keep track of the pending remote writes. The pending remote updates are organized into separate linked lists based on their target remote thread, with each node containing the shared address and value of the update.

Since the number of cache blocks is limited, all cache blocks may be in use when a cache miss occurs, and thus an existing block must be spilled to accommodate the new request. We use a standard clock algorithm to pick from the ready cache blocks: a cycling index searches for unused blocks when a block is needed, and it marks used blocks as “unused” during its sweep. Each hit to a cache block marks the block as “used”, thus providing a cache replacement policy that approximates the least recently used (LRU) algorithm. Note that a block in the “pending” state is never marked by the clock algorithm as “unused”, as a pending read operation has been guaranteed that the memory will be resident when the read’s “sync” operation is called (and on some platforms, the network hardware may be asynchronously writing to the block). In the common case where an asynchronous read hits the cache, the read is serviced during the “init” section of the read, and the “sync” is a no-op: this allows a resident cache block to be evicted even if there are outstanding reads against it.

3.2 Maintaining Cache Coherence

Because our caching system takes the local knowledge only approach, maintaining cache coherence is as simple as having each thread flush its cache at each synchronization point, which consists of the language-defined synchronization operations as well as strict memory accesses. UPC currently supports the following synchronization primitives: barrier statements to divide a program into synchronization phases, post-wait synchronization for producer-consumer dependencies, memory fences equivalent to a null strict reference, and finally lock based synchronization associated with lock acquires and releases. While invalidating all data in the cache is a conservative implementation, the approach eliminates the overhead of coherence messages as well as the cost of bookkeeping of the possible owners for each shared variables. Furthermore, cache invalidation should be a rare event, thanks to the use of a relaxed memory consistency model. Since the system issues non-blocking puts for remote writes, at synchronization points it must also flush the update queue to ensure that all of the previous write operations have been globally performed. Finally, the synchronization operations themselves must be completed before any later shared memory accesses or synchronization operations can begin to execute. With this set of behaviors, our implementations satisfies the constraints specified in our new definition of the UPC consistency model in Section 2.3.

3.3 Handling Relaxed Variable Reads

For every shared memory read where the source address *addr* is located on a remote thread, the caching system first checks if the desired value is present in the thread’s local cache. If this results in a cache hit, the system directly returns the cached value to the user. Otherwise, it finds a cache entry for replacement and initiates a non-blocking network call to fetch the entire cache block that *addr* belongs to. To effectively hide communication latencies, control is immediately transferred back to the user code without waiting for the completion of the network call. The cache block is guaranteed to be available after the compiler has issued a `sync()` on *addr*, at which point all subsequent accesses to shared addresses within the block will be serviced as cache hits.

3.4 Handling Relaxed Variable Writes

Since most parallel programs follow the "owner-computes" rule by fetching remote data as input and store the computation results locally, remote updates should occur relatively infrequently compared to remote reads. As a result, for relaxed remote writes our system currently adopts the easy-to-implement write-through policy; a non-blocking message send is immediately issued to the remote destination, and control is then transferred back to the user code. To ensure the eventual completion of the remote write operation, a handle to it is created and appended to the thread's update queue, which record all of the pending outgoing updates.

One additional problem posed by asynchronous writes is that to preserve single-processor data dependency, the system must guarantee that subsequent reads from this thread to the same memory location will reflect the update. If the cache block in question is already present in the cache, we need to perform the update on the block so it will now hold the new value. Even if the stale value is not cached, additional actions are still required to maintain coherence due to *false sharing*: since the remote relaxed writes are implemented as non-blocking calls, due to network reordering we do not know when the update messages will arrive in the destination. If the same thread issues a relaxed read to another memory location in the same cache block (such sequence still obeys the synchronization model as the two operations access different memory locations), it could inadvertently see the stale value if the update is still in progress. To resolve this inconsistency, whenever a read miss occurs, the thread must first examine its update queue for conflicting writes to the same cache block, and wait for their completion before it can proceed with the remote $get()$. Similarly, before a write is issued, it must check to see if the cache block it would write to is in the "pending" state, and if so it must wait for the block read to complete before issuing the write message. Since the cache block lookup is needed anyway (in order to write to the block contents if they are resident), it does not incur any significant overhead.

3.5 Handling Bulk Accesses

Bulk shared memory loads ($upc_memget()$) are not cached. Programmer-initiated bulk transfers will generally be done as part of a conscious optimization strategy, in which the data will be copied into the calling thread's private memory area and then accessed efficiently via regular C pointers avoiding the overhead of global pointers and cache lookup logic, and making caching superfluous.

At present, the UPC compiler guarantees that any earlier writes that may conflict with a bulk read will already have been completed by the time the bulk read is initiated. The read can thus completely bypass the caching mechanism, with no additional overhead. Note that this may result in a bulk read seeing a more recent remote value than is stored in the cache, and a later, small relaxed accesses may see the older cached value. This can only occur if a conflicting write to the shared variable has occurred, which violates the UPC synchronization model, and so the implementation is not required to provide a coherent picture of the memory until the next synchronization point.

Bulk shared memory updates do require some interfacing with the caching mechanism. Later reads from the same thread must see the written values, and so any cache blocks that overlap with the write must be found and either flushed or updated with the written values. Also, the write must be stored in the update queue, so that any cache misses that cause a conflicting block to be loaded can first sync the bulk write, so that they see its changes. Again, this is required because of "false sharing" even though the compiler guarantees that particular data accesses do not conflict.

4 System Optimizations

In this section, we describe several optimizations under consideration for our caching system. While not every optimizations listed here have been implemented due to time constraints, they will all be included in the final release of our system.

4.1 Overlapping Communication with Computation

As mentioned in Section 3, when a cache miss occurs the system issues a non-blocking $get()$ to fetch the remote value, with the hope that the latency associated with the network transfer can hidden through communication and computation overlapping. A similar approach is taken for remote writes by not blocking for their completion until the next synchronization point. Maximizing the benefit of the optimization, however, requires the cooperation of the compiler, as the use of non-blocking operations to serve cache misses will lose its effectiveness if the compiler needs to use the remote value immediately. Fortunately, the open64 compiler framework [23], which the Berkeley UPC compiler is based on, already implements a pointer-based prefetching algorithm described in [21], which can be adopted to prefetch remote data. To separate the initiation and completion of the non-blocking accesses, the compiler pushes the $init\_get()$ backward to the prefetch label, while the $sync\_get()$ remains at the original location of the remote access. A more aggressive optimization can be applied for remote updates; since our memory consistency model allows the reordering of relaxed writes with no intervening synchronization operations, we can effectively perform message pipelining for all outstanding writes, blocking for their completion only when reaching a later synchronization point.

4.2 Buffering Remote Writes

As discussed in the previous section, our caching system currently uses a write-through protocol because we believe that remote updates are unlikely to be the performance bottleneck for most programs. To fully exploit the flexibility offered by our memory consistency model, however, we could adopt a write-back policy to further reduce the communication overhead. Under this protocol, when encountering a remote relaxed write the system simply stores the operation in a message packing buffer (with a different buffer available for each target node). The writes to this buffer can be arranged to begin in the middle, with each message prepending the target address and message length to the head, and the message data to the end. Following writes that are contiguous with the last write can simply update the length field rather than adding a separate address/length field, thus providing an automatic message packing mechanism. The message packing buffer is flushed when the buffer is full, or at the next synchronization point, where all previous write operations must have been completed. The message can be sent to the target thread as a single block, which the remote thread can then scatter into the correct target destinations. While this strategy delays the propagation of writes (and thus potentially delays subsequent strict accesses or barriers), it reduces the number of messages required through message coalescing. Since on many networks the fixed software overhead and latency of a message transfer are significant, a message of even several hundred bytes or more can be almost as efficient as a single 8 byte transfer [13], and thus message packing can be very effective at lowering total software overhead involved in message transfer.

It is interesting to note that this write buffering could actually include strict writes in the buffering. So long as a thread's strict writes are made visible in program order, they can be delayed, so in the absence of other remote operations (or strict accesses to shared data with local affinity), strict writes can be coalesced in the write packing buffer, so long as it is guaranteed that when the buffer is actually sent, the writes it contains are unpacked in their original order. This could only work if the strict writes were all to a single thread, however, as it would be difficult to guarantee the ordering of unpacking on different threads. For the same reason, the buffers for other threads would need to be sent and completed before the one containing the strict write(s). Finally, relaxed writes to shared data with local affinity would have to result in the message buffer being first sent and synchronized (otherwise other threads' could potentially see the later relaxed writes before the strict writes were visible, which would be a violation of our proposed UPC memory model). This requirement would add a check for every local shared write, which might make this optimization too expensive, however.

4.3 Performing conflict checking in the runtime

Currently our UPC compiler guarantees the runtime that any potentially conflicting reads/writes from a single thread will be handled via the compiler inserting a sync of a previous (potentially) conflicting operation before the initialization of the other operation. However, this may result in suboptimal behavior, since aliasing and other problems may require the compiler to be fairly conservative in its assumptions. It may be possible to achieve some speedup by having the runtime system do checking for conflicts at runtime (when perfect information is available), thus freeing the compiler to perform more aggressive communication optimizations.

The caching system here could interact favorably with such an analysis. The write packing buffer scheme just mentioned, for instance, would trivially handle write after write conflicts, so long as the packed message was unpacked in program order. In the absence of packing each write would otherwise need some way to lookup whether any conflicting earlier writes were still pending.

4.4 Optimizing Lock Based Synchronization

The current UPC memory model states that both unlock and lock operations will implicitly perform a null strict reference, so that the cache and the update queue must both be flushed in either case. This restriction, however, is overly conservative as it fails to exploit the semantics of mutual exclusion implied by the lock operations. In particular, if we assume that a program uses only lock based synchronization, then writes performed by a thread before a lock release need to be globally propagated only after another thread has acquired the lock that was released. This assumption is valid because accesses made by a thread inside a critical section are not observable by other threads until the thread has released the lock guarding the critical section. We can therefore further relax our caching coherence protocol by adopting the release consistency model [6] for lock based synchronization, which specifies the following constraints:

	Local Read	Remote Read	Cache Hit
Time (ns)	20	27000	350

Table 1. Software Overhead of the Cache Operations

1. (acquire $\longrightarrow$ all) All shared accesses must wait before the preceding (in program order) lock acquires are completed
2. (all $\longrightarrow$ release) All shared accesses must be globally performed before any subsequent lock releases could proceed.
3. (acquire $\longleftrightarrow$ release) Acquires and releases must be executed following program order

This relaxed definition allows us to implement asymmetric cache coherence actions for releases and acquires. When a thread releases a lock it must flush its update queue to ensure that the results of its writes since the last release are propagated; similarly when a thread acquires a lock, it must flush its cache to ensure that it will receive the updates from threads that had previously held the lock. We choose not to implement this mechanism as the use of locks appears to be almost non-existent in the UPC programs we have encountered so far; this optimization, however, can certainly be highly beneficial to applications with frequent lock operations, as it can improve cache performance dramatically by eliminating unnecessary cache invalidations. Furthermore, the same technique can be extended to post-wait synchronization, for which we treat post as a release and wait as an acquire operation.

5 Performance Results

5.1 Experimental Setup

We implemented our software caching mechanism inside the runtime system of the Berkeley UPC compiler. Experiments were performed on the Alvarez supercomputer at the Lawrence Berkeley National Laboratory [2], a 80-node two-way Linux cluster with a Myrinet interconnect.

5.2 Software Overhead of the Caching System

Table 1 presents the costs of various basic cache operations as reported by some simple microbenchmarks. All figures are for 8 byte reads. The cache hit figure is for the optimal situation, where the first cache descriptor in the linked list off the hashtable is the match. We believe further improvements in the cache hit timing may be possible via optimizing the hashing logic, but even with the present results, it is clear that an access to a cache block is several orders of magnitude faster than a remote access, at least for the Myrinet network used here. Remote read performance is within the same order of magnitude for most other high performance networks available today, indicating that caching has clear potential for application speedup.

5.3 Application benchmarks

Next we measure the effectiveness of our caching system on a naive matrix multiplication benchmark, which should benefit from caching immensely due to its regular fine-grained access pattern. The application was taken from [12], with a matrix size of $256 * 256$ with 64-bit doubles. For comparison purposes, we also measure the running time of a blocked matrix multiply that uses bulk prefetching to fetch the remote values; this serves as the upper bound on the maximal performance boost that our caching implementation could achieve. As the figure shows, caching achieves a significant improvement over the naive algorithm: on 16 processors, the running time drops from 34 seconds to only .58 seconds, while the perfectly prefetching bulk version takes .31 seconds.

Similarly, a simple GUPS benchmark (which performs random accesses to memory locations) shows significant speedup from caching, though not by as large a factor as for the matrix multiply. This is to be expected, given that the locality and density characteristics of GUPS are significantly less than that of matrix multiplication.

6 Bulk Prefetching for Regular Accesses

While caching is effective in reducing the overhead associated with a shared memory programming style, it's unlikely to match the performance of equivalent coarse-grained programs that use bulk memory transfers. For fine-grained programs

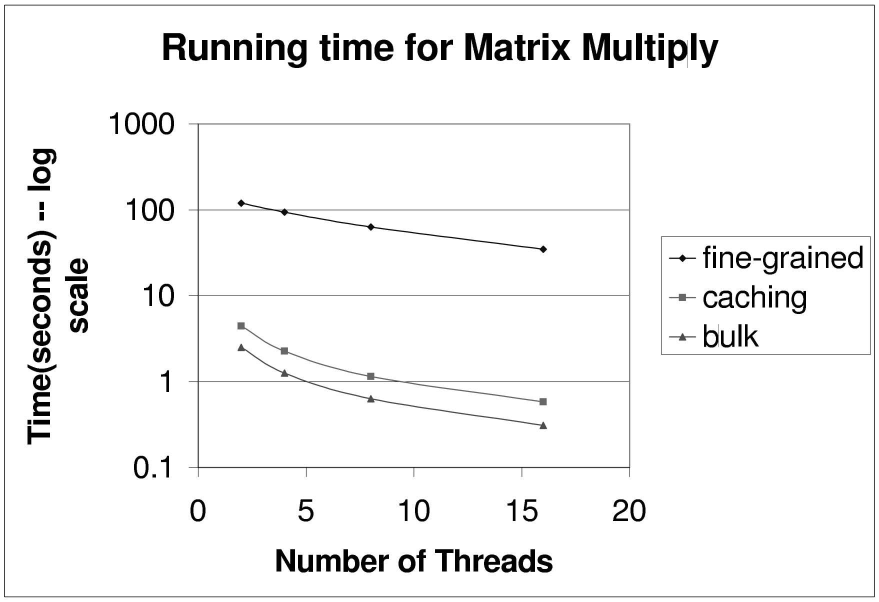


Figure 6. Running time for Matrix Multiply

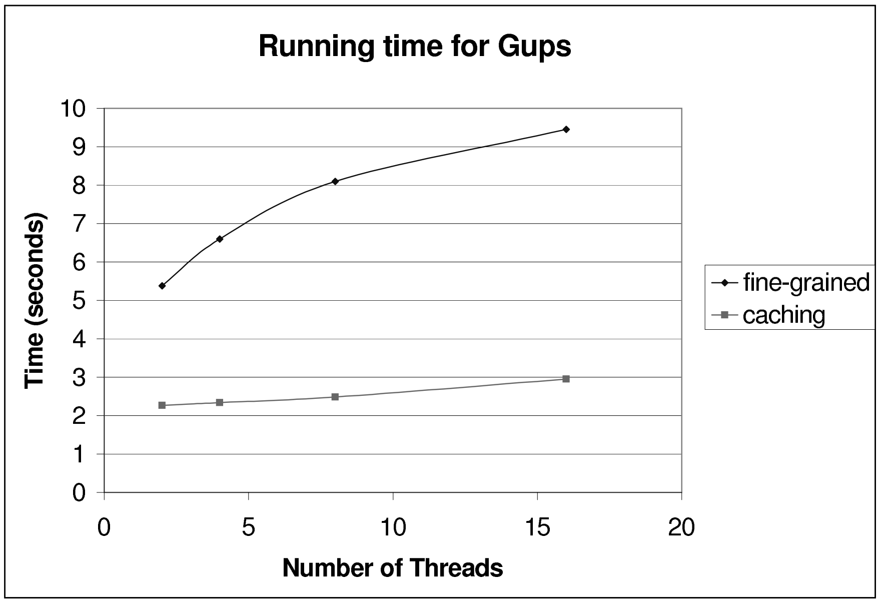


Figure 7. Running time for Gups

with regular access patterns, however, we can further reduce this gap by performing bulk prefetching to automatically convert small message communication into bulk memory operations. Here we discuss our preliminary experiences with such a transformation in another SPMD global address space language, Titanium.

6.1 Titanium and its foreach loops

Titanium[15] is a dialect of Java designed for high-performance parallel computing using a SPMD execution model. Titanium inherits many features of Java, one of which is the Java memory consistency model. The Java memory consistency model is defined in [14]:

1. **Locally sequentially consistent:** For a single processor, all reads and writes to a given memory location must appear to occur in exactly the order specified.
2. **Globally consistent at synchronization events:** At a global synchronization event, such as a barrier, all processors must agree on the values of all the variables. At a non-global synchronization event, such as entry into a critical section, the processor must see all previous updates made using that synchronization event.

Our transformation targets the builtin `foreach` loops in Titanium. A foreach loop has the form of `foreach (p in D) S`, where the iteration space D is an arbitrary subset of positive integers, and S any sequence of statements. The loop's semantics specifies that the body, S, be executed $|D|$ times with p bound to each element of D in each iteration. However, the order in which p iterates through D is unspecified. The current implementation of domain is a list of rectdomains, and from now on we will assume that the iterations are over rectdomains. Extensive analysis and optimization for foreach loop have been added to the Titanium compiler [24]; the relevant analyses for our purposes are dominator analysis for foreach loops and MIVE. The classical dominator analysis is used to determine the following:

1. A foreach loop is a full domain loop or a partial domain loop. A full domain loop executes every iteration and cannot be cut short except by a fatal error.
2. An array access appears on every iteration of the loop.

`MIVE` stands for "Map from Induction Variables to Expressions." MIVE is used to find array accesses inside of foreach loops whose address calculation can be strength reduced. Instead of computing the address for an array access from scratch every time, the address for the current iteration can be computed from the address for the previous iteration by adding a pointer increment that is loop invariant. This means that the set of addresses that an array access in a foreach loop will touch can be computed by the rectdomain that we are iterating over and the loop invariant pointer increments.

During compilation, Titanium code is translated into C code, and then the generated C code is compiled by the C compiler. An array access in Titanium code can be translated into a global pointer access or local pointer access in the C code. Global pointer access handles the general case, where the data can be remote or local. If an array access is known to be local at compile time, then it is translated into a local pointer access. Figure 8 shows the generated code for the two different accesses.

6.2 Fine-grained vs. Coarse-grained

For a programmer that is used to writing sequential programs, it is likely that she would write her first parallel program in a fine-grained fashion, because it is most similar to the sequential program. Global Address Space languages such as Titanium and UPC encourage the fine-grained programming style. In Titanium, the interface for a remote access is identical to a local access, although the remote access is several orders of magnitude slower than the local access. A fine-grained program is likely to have many small remote accesses, each costing a blocking round trip. The alternative is to write the program in a coarse-grained fashion. The application programmer would write code that explicitly transfers remote data into local buffers using bulk operations. The data retrieved is used during subsequent computation. Figure 9 illustrates the difference between a fine-grained program and a coarse-grained program.

A fine-grained program is easier to write than a coarse-grained program. But there is a large performance gap between the fine-grained program and the coarse-grained program. There are two reasons for this performance gap. They are the following:

1. In Figure 9, the fine-grained program has a network round trip for each read of $A[i]$, while the coarse-grained program has only one network round trip for the entire loop. For the coarse-grained program, the remote data is retrieved using one array copy before the loop.

```
local pointer access      global pointer access
value=*ptr                if (ptr is local){
                             value=*ptr
                          }
                          else{
                             value=retrieve *ptr from remote location
                          }
```

Figure 8. Code generation for local pointer access and global pointer access

```
                          A is a remote array

Fine-grained Version              Coarse-grained Version
sum = 0;                          sum = 0;
foreach (p in A.domain()){        double [1d] local copyA;
            sum += A[p];          copyA = new double[A.domain()];
}                                 copyA.copy(A);
                                  foreach (p in copyA.domain()){
                                              sum += copyA[p];
                                  }
```

Figure 9. Titanium code for fine-grained program and coarse-grained program

2. We also notice that copyA is declared as local in the coarse-grained program. This allows the compiler to generate local pointer access code for each array access of $copyA[i]$. While in the fine-grained version, the array access A[i] is translated into global array access, because A is not known to be local statically. In this case, A is a remote array.

The goal of prefetching is to narrow the gap between fine-grained programs and coarse-grained programs. Specifically, the rest of this section will concentrate on array prefetching for array accesses inside of a foreach loop. In the case of Figure 9, we would like to prefetch the data for A[i].

6.3 Implementation

In compile time, we identify strength reduced array accesses inside of foreach loops as potential prefetch candidates. The goal is to find candidates whose prefetched data will be used during every iteration of the foreach loop. The two criteria are the following:

1. The foreach loop is a full domain loop.

2. The array access appears on every iteration of the loop.

Prefetching effectively hoists reads inside of a loop to the header. We need to check if such a move would violate the Titanium memory model. As stated in section 6.1, memory needs to be globally consistent at synchronization points, this includes barriers and locks. For our initial implementation, we make sure that such synchronization operations do not get called during the execution of the foreach loop, so the prefetched data remain valid during all iterations of the loop.

At this point, we have the set of array accesses that we would like to prefetch for. During code emission (to C code), codes for prefetching for those array accesses are inserted in the header of their respective loops. Since the array accesses are strength reduced, the elements of the array that are needed in the loop can be computed from the rectdomain that we are iterating over and the loop invariant pointer increments found by `MIVE`. Prefetched data are stored in local buffers, so array accesses inside of the loop is known to be local at compile time. We proceed to change those global pointer references into local pointer references.

During runtime, the prefetch code is called during the setup of the foreach loop. Three situations can occur:

1. The data is already in the buffer due to previous prefetch operations. We can simply return a pointer to the data.

2. The data is actually addressable from the executing thread. This can happen when the executing thread owns the data or a processor on the same node owns the data. In this case, the data is actually local, we simply return a pointer to the data.

3. The data is remote. We call a non-blocking bulk read operation with an explicit counter to retrieve the data from the remote location. If we have multiple array accesses to be prefetched for the same loop, the non-blocking operations are issued one after the other. Communications for different prefetch operations are overlapped. At the end of the loop setup code, we synchronize on the counter to wait for the bulk read operations to complete. This allows us to overlap the bulk read with the loop setup computation.

Although we make sure that synchronization operations do not get called during the execution of the foreach loop, there may be synchronization operations outside of the loop. To obey the memory consistency model, we flush the buffers at those points.

The first rule of the Titanium memory consistency model requires memory to be locally sequential consistent. If the executing thread writes to the prefetched data, we need to resolve those conflicts by merging in the changes. If the data to be modified is actually local, then such a conflict does not exist, since there is only one copy of the data. When the data is remote, we have a copy of the data in a local buffer. In this case, we need to merge in the changes into the local buffer, so subsequent reads from the executing thread would see its own writes. Runtime checks are inserted into remote writes and array copies to handle this case.

6.4 Performance

The benchmarks presented below were run on Seaborg (IBM SP) [1] with the following configuration, two nodes with four processors on each node. Seaborg uses a SP Switch2 switch. The CPU type is 375 MHz Power 3+.

	Fine-grained	Bulk Prefetch	Coarse-grained
Shark and Fish (seconds)	376	221 (1.7x)	223 (1.7x)
Matrix Vector Multiply (ms)	27354	10 (2735x)	10 (2735x)

Table 2. Performance of the Titanium Benchmarks. The parenthesized numbers reflect the speedup over the fine-grained version.

	Message Coalescing	Local Pointer Access	Total Speedup
Shark and Fish (seconds)	152	2	154
Matrix Vector Multiply (ms)	27328	16	27344

Table 3. Speedup due to message coalescing and local pointer access

6.4.1 Shark and Fish

We first tested our techniques on a shark and fish particle simulation program. At every time step, the forces on each particle are summed up due to all of the other particles. Particles are distributed evenly among processors and have immutable type, so that the entire content of the particle object would be transmitted over the network on a remote pointer access instead of a pointer to the content. The benchmark was run with a problem size of 1000 particles. Table 2 illustrates the performance difference between the three versions of the code: fine-grained, prefetch, and coarse-grained.

The prefetch version has comparable running time as the bulk version. Comparing to the fine-grained version, the prefetch version obtains a speedup of 1.7x. The prefetch version reduces the number of network roundtrips by a factor of $N/numberofprocessors$, where N is the problem size. It has the same number of round trips as the bulk version.

6.4.2 Dense Matrix Vector Multiply

The dense matrix vector multiply computes y=A*x, with each processor computing for its section of the y vector. Communication is required for retrieving parts of the x vector owned by other processors. A 1D row layout is used for matrix A. The benchmark was run with a matrix of size 1024x1024 doubles. Table 2 illustrates the performance difference between the three versions of the code: fine-grained, prefetch, and coarse-grained.

The prefetch version has comparable running time as the bulk version. In comparison to the fine-grained version, the prefetch version obtains a large speedup of 2735x. This significant speedup is due to the reduction in network round trips. The reduction factor is $(N/numberofprocessors)^2$, where N is the dimension of the matrix.

Compared to the fine-grained version, prefetching offers the following benefits:

1. **Message coalescing:** Instead of doing a round trip for each remote array access, elements of the array that will be used during the foreach loop is retrieved via one bulk read operation.

2. **Local pointer access:** For array accesses that have their data prefetched, it is known at compile time that the data will be local during the execution of the foreach loop. Therefore, we can safely change those global pointer accesses to local pointer accesses. As stated in section 6.1, global pointer access handles the general case, where the data can be remote or local. During runtime, a global pointer access checks for the location of the data, and follows different code paths for local and remote data. By statically changing those global array accesses to local array accesses, we bypass the check for data location. The translated C code is a simple pointer dereference.

We quantified the speedups in the previous section into message coalescing and local pointer access. Table 3 displays this data, and it is clear that message coalescing is the dominant factor for the speedup.

7 Conclusion and Future Work

In this paper we have described the design and implementation of a software-based caching system for UPC. While our prototype system is immature and could be improved in many ways, preliminary performance results suggest that it can be beneficial in reducing the average cost associated with small message reads and writes. Another contribution made in this paper is that we identified the problems with the current UPC memory consistency model being overly restrictive, and propose a new and more flexible definition. We believe that our proposed memory model can be very helpful for UPC performance, as it permits more aggressive compiler and runtime optimizations, including caching, that can not be implemented efficiently under the current model. Finally to further bridge the gap between fine-grained and coarse-grained applications, we discuss compiler techniques that could automatically transforms array accesses in loops into equivalent bulk prefetching operations. We have measured the effectiveness of this optimization in Titanium, and the results suggest it is highly effective for applications with regular array access patterns.

Much work still remains to be done. First, our prototype system could benefit from extensive performance tuning. We would also like to use the cache system to execute more standard benchmarks to more accurately evaluate its performance. Another important work is to port the caching system to other global address space languages such Titanium so that they could also benefit from the effects of caching. Finally we are also experimenting different optimization techniques for application with irregular access patterns, for which bulk prefetching would no longer apply.

References

[1] IBM SP. http://hpcf.nersc.gov/computers/SP.

[2] NERSC Alvarez Cluster. http://www.nersc.gov/alvarez.

[3] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE computing*, 1996.

[4] S. V. Adve and M. D. Hill. Weak ordering—A new definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 2–14, 1990.

[5] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An evaluation of current high-performance networks. In *the 17th International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[6] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, 1991.

[7] W. Chen, D. Bonachea, J. Duell, P. Husband, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC Compiler. In *Proceedings of ICS 2003*, 2003.

[8] A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pages 106–117, 1994.

[9] E. Darnell and K. Kennedy. Cache coherence using local knowledge. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*.

[10] T. El-Ghazawi and F. Cantonnet. UPC performance and potential: A NPB experimental study. In *Supercomputing2002 (SC2002)*, 2002.

[11] T. El-Ghazawi, W. Carlson, and J. Draper. *UPC specification*, 2003. http://www.gwu.edu/ upc/documentation.html.

[12] T. El-Ghazawi and S. Chauvin. UPC benchmarking issues. In *30th IEEE International Conference on Parallel Processing (ICPP01)*, 2001.

[13] T. Fahringer and E. Mehofer. Problem and machine sensitive communication optimization. In *International Conference on Supercomputing*, pages 117–124, 1998.

[14] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*, second edition, 2000.

[15] P. Hilfinger et al. Titanium language reference manual. Technical Report CSD-01-1163, University of California, Berkeley, 2001.

[16] L. Iftode and J. P. Singh. Shared virtual memory: Progress and challenges. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):498–507, 1999.

[17] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Jorunal of Parallel and Distributed Computing*, 1996.

[18] W. Kuchera and C. Wallace. Illustrative test cases for the upc memory model. Technical Report 03-02, Michigan Technological University, 2003.

[19] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers*, 1976.

[20] J. Lee and D. Padua. Hiding relaxed memory consistency with compilers. In *proceedings of The IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[21] C. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. In *Architectural Support for Programming Languages and Operating Systems*, pages 222–233, 1996.

[22] S. Midkiff and D. Padua. Issues in the compile-time optimization of parallel programs. In *Proceedings of the 1990 International Conference on Parallel Processing*, 1990.

[23] open64. http://open64.sourceforge.net.

[24] G. Pike. *Reordering and Storage Optimizations for Scientific Programs*. PhD thesis, U.C. Berkeley, 2002.

[25] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 1988.